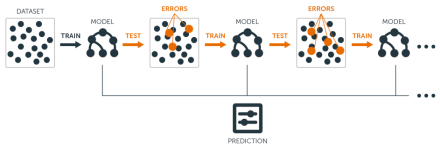
Trees and ensembles
Part 1: Decision trees

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

We saw decision trees for the first time in the very first lectures, and we've seen them a few more times since. But we never actually discussed how to train them. In this lecture, we'll look at the details of that.

---

gradient boosted decision trees



source: https://blog.bigml.com/2017/03/14/introduction-to-boosted-trees/

Decision trees by themselves are not a very popular model in modern machine learning. They are quick to train, but they can be prone to overfitting, and regularizing them hurts performance a lot.

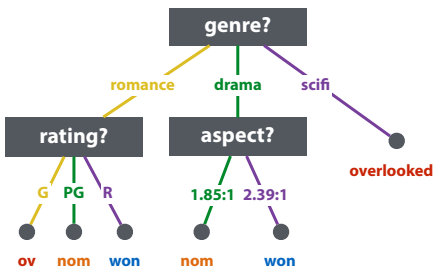
The main setting in which trees are used are in **ensembles**, a model that combines a lot of other models in order to arrive at a prediction. Spefically, the method of gradient boosted decision trees, is a very popular approach for achieving high performance with relatively little effort. In this lecture, we will build this picture up step by step: we will first discuss decision and regression trees in the first two videos, and then different ways to create model ensembles in the last two videos.

rating	genre	aspect ratio	outcome
PG	scifi	1.85:1	overlooked
G	drama	1.85:1	won
G	romance	1.85:1	nominated
R	drama	1.85:1	nominated
G	drama	2.39:1	nominated
G	romance	2.39:1	nominated
R	romance	1.85:1	won
PG	drama	2.39:1	won
PG	scifi	1.85:1	overlooked
G	scifi	2.39:1	overlooked

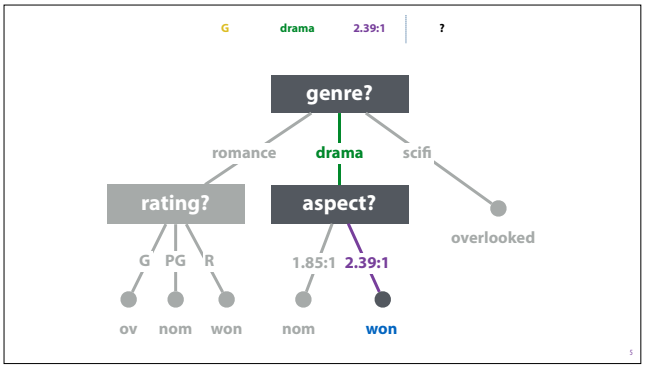
Decision trees in their simplest form work on data with *categorical* features. We'll use this dataset as a running example: each instance (row) is a movie, and the target class is to predict whether a movie **won** an oscar, was merely **nominated**, or **overlooked**.

---

decision tree



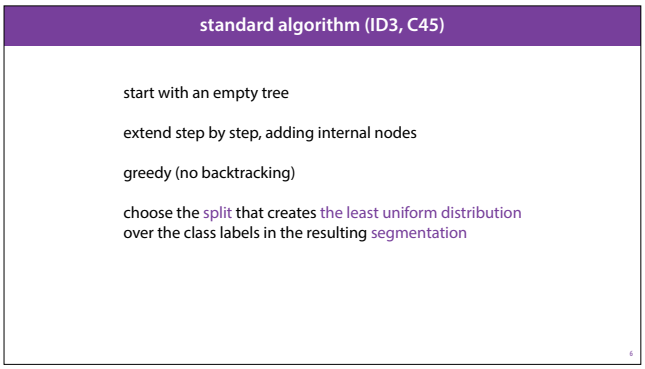
This is what a trained decision tree might look like. Each internal node asks the value of a particular feature, and sends the instance to one of its children dependending on the value of that feature.



To classifiy an instance by this decision tree, we start at the root (the node at the top), and work our way down by answering the question in the node.

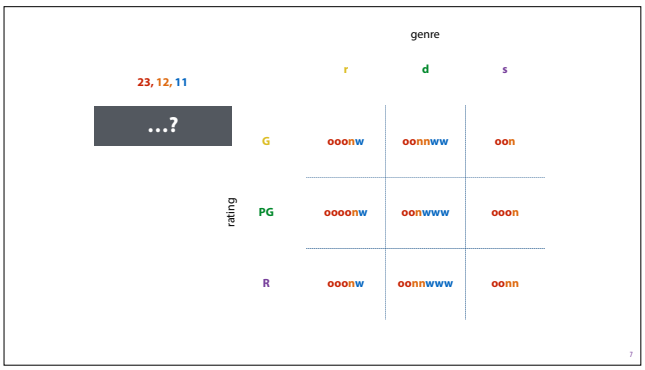
If we see a movie with a G rating, the genre drama, and a 2.39:1 aspect ratio, we follow the tree to the highlighted leaf node and label the example as **won** (i.e we predict that this movie will win an oscar).

So, given the space of all possible trees, how do we find one that fits our data well?



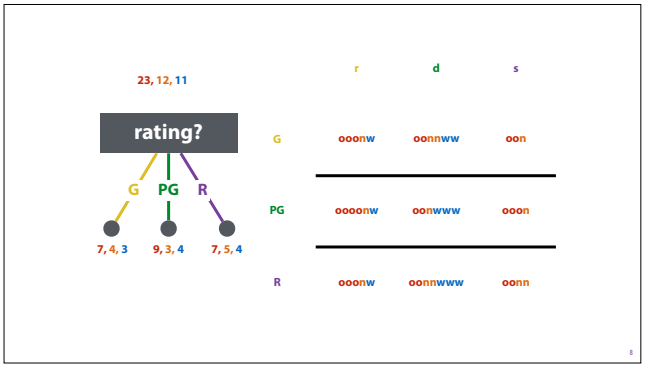
This is how the basic algorithm is set up. We start with an empty tree, and add one node at a time. We don't backtrack: once a node is added it stays in the tree, and we keep adding nodes until we can add no more.

The node we add at any given moment, is the node that creates the **least uniform distribution** on the classes, after the split. Let's look at an example to see how this works.

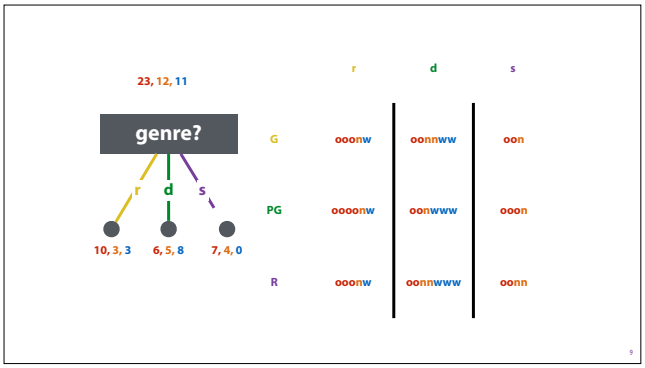


Here we've plotted our data for two features (ignoring the third for the moment). We're choosing the root node of our tree

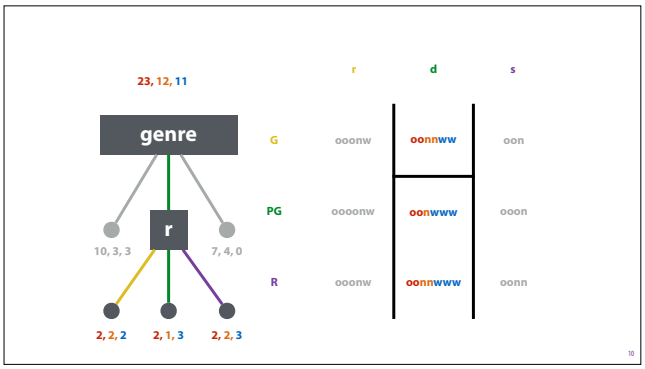
---



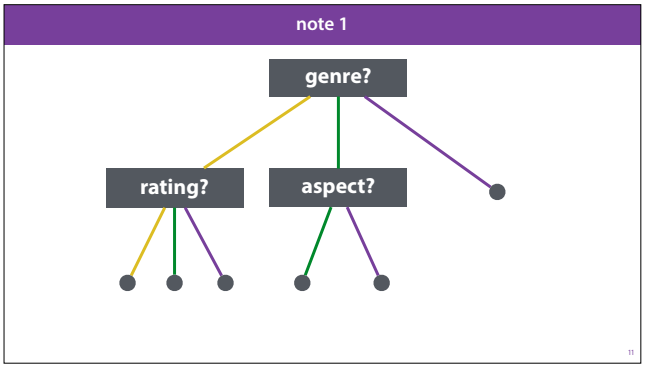
If we split by rating, we get three segments (the three rows on the right). Tallying up the proportions of each class, we see that the proportions of the segments are not that different from the proportions in the whole. In other words, knowing the value of the rating doesn't doesn;doesn't change the information we have about the class very much.



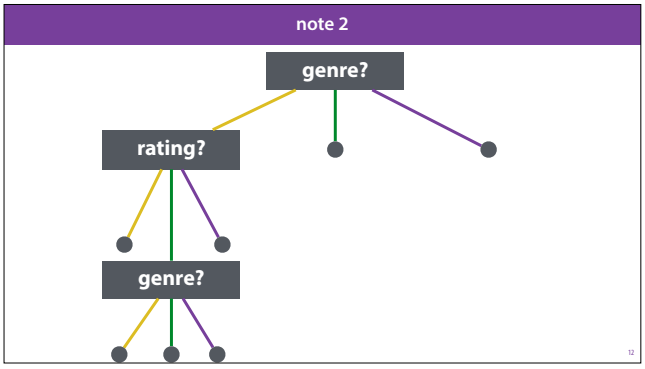
On the other hand, if we split by genre we see that the resulting distributions are much more different: knowing that a movie has the genre **scifi**, allows us to say with near certainty that it won't win an oscar.



If we split by genre and then by rating, we get this segmentation of the instance space. Each of these regions, corresponding to a leaf node is called a **segment**.



We choose a separate split for each node we extend. For each of the three children of the genre node, we may choose different features to split on.



There's no use (with categorical features) in splitting on the same feature twice. Every instance that encounters the lower genre node will have the yellow genre, so we're not splitting the data at all.

stop conditions

When the maximum depth has been reached.
output label: majority class

When all feature values are the same
output label: majority class

When all class values are the same
output label: left over class

13

The maximum depth is an optional hyperparameter. We can also train without a maximum depth and rely only on the other two stop conditions.

---

how do we define "uniformity"

14

In order to make this into a proper algorithm we need to make this more precise. The best feature to split on is the one that creates (averaged over all child nodes) the most non-uniform class distribution in the resulting segment.

How do we measure the non-uniformity of a distribution? It's straightforward for two classes (the further from 50/50 the less uniform), but for more than two classes, it's not so clear cut. Here we see two distributions. In the first, the proportion of the red class is bigger than in the second, but in the second the remainder is divided up between blue and orange in a more non uniform way. Which of these two distributions is less uniform?

---

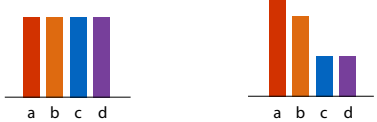
entropy

p(X): data source

$$H(p) = \sum_{x \in X} p(x) L(x)$$
$$= -\sum_{x \in X} p(x) \log p(x)$$

15

To answer this, we need to look back to the first probability lecture, where we encountered *Entropy*.

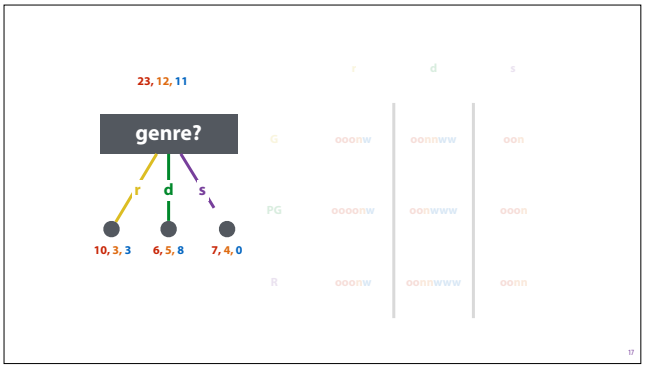


H(p) = 2 bits

H(p) = 1.75 bits

16

The more uniform our distribution is (the more unsure we are) the higher the entropy.



So we can use entropy to establish how uneven a class distribution is, and the more uneven, the better we like the split. But to evaluate this split here, we need to look at four different distributions: the three after the split and the one before (remember, we are evaluating all possible splits over the whole tree, so the incoming distribution may differ between candidates).

---

conditional entropy

$$p(\text{Outcome} = \text{won} \mid \text{Genre} = \text{drama})$$

$$H(O \mid G = d) = -\sum_{o \in \{o,n,w\}} p(o \mid d) \log p(o \mid d)$$

$$H(O \mid G) = \mathbb{E}_g H(O \mid G = g) = \sum_g p(g) H(O \mid G = g)$$

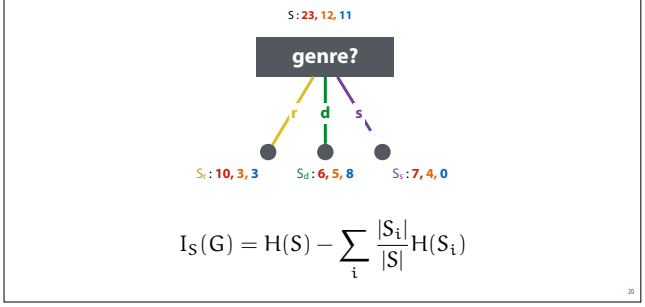
To apply this to the multiple children that a split creates, we can use **conditional entropy**. Conditional entropy is just the entropy of a conditional distribution, summed over all values of the conditional, weighted by the marginal probability of that value.

---

information gain of G

$$I_O(G) = H(O) - H(O \mid G)$$

The **information gain** measures how much knowing the value of G decreases the entropy of O (i.e. increases what we know about O).



$$I_S(G) = H(S) - \sum_i \frac{|S_i|}{|S|} H(S_i)$$

In practice that gives us this formula, for the information gain of a split. If the set of instances before the split is S, and the split gives us subsets S_i, the information gain is the entropy of S minus the sum of the entropies of the split sets, each weighted by the proportion of instances of S contained in S_i.

When we compute the entropy of a set like S, we just use the relative frequencies to estimate the probabilities. For instance, we estimate the probability of the **red class** in S as **23**/(**23**+**12**+**11**).

algorithm

start with a single unlabeled leaf

loop until no unlabeled leaves:

  for each unlabeled leaf l with segment S:

    if stop condition: label majority class of S

    else: split l on feature F with highest gain $I_S(F)$

Trees and ensembles

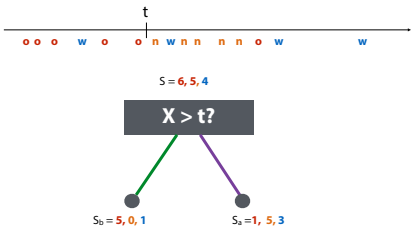
Part 2: Numeric features and targets

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

---

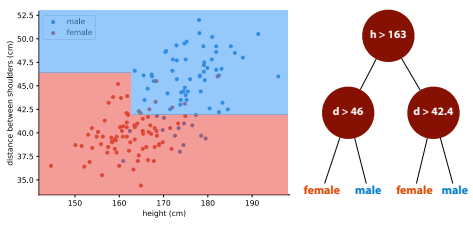
numeric features

split at *threshold* t

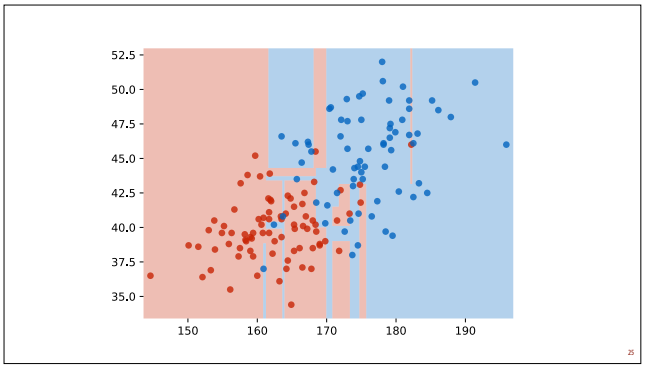


If our dataset contains numeric features, we can deal with this by choosing a threshold t, the node splitting on a numeric features splits the segment in two: the instance for which the feature is lower than t go to one child, and the instances for which the features is higher go to the other.

To compute the optimal threshold we only need to look at numeric values halfway between two instances with a different class. We compute the information gain for each and choose the threshold which provides the highest information gain.

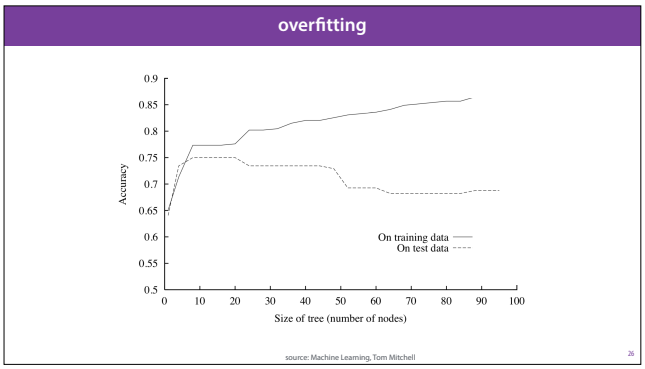


We saw a classifier with numeric features in the opening lecture.



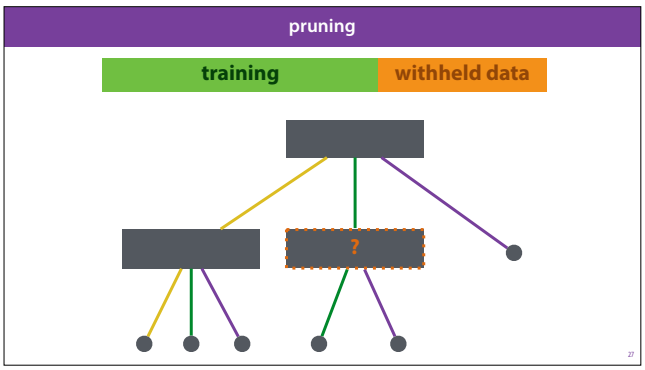
When training an actual decision tree on these two numeric features, we saw quite a complex decision boundary emerge.

This is possible because with numeric features it *does* make sense to split twice on the same feature; we just have to split on a different threshold each time.



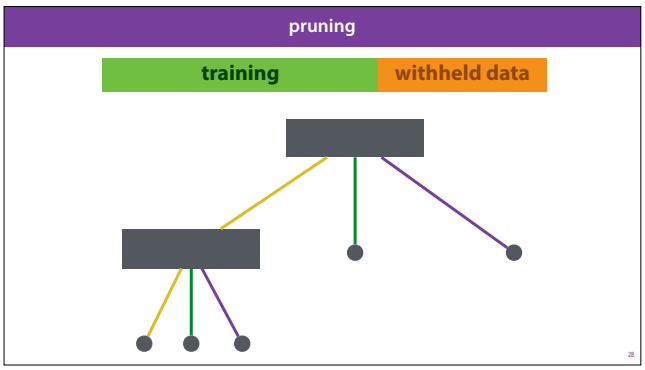
Which brings us to the problem of overfitting. The larger the tree grows, the more likely it is to overfit the data. For this plot the size of the tree is limited to a particular maximum. As the maximum grows, the training accuracy increases, but the test accuracy decreases. A clear sign that the model is overfitting.

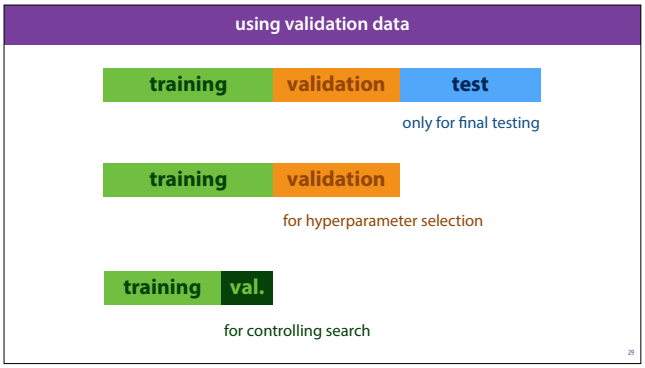
Source: Machine Learning, Tom Mitchell



To reduce overfitting, we can **prune** a tree.

After training the full (likely overfitting) tree, we work backwards from the leaves. For each leaf, we check (on withheld data) whether the tree classifier better with the leaf or without. If it works better without, we remove the node.We keep pruning leaves until the performance stops improving.

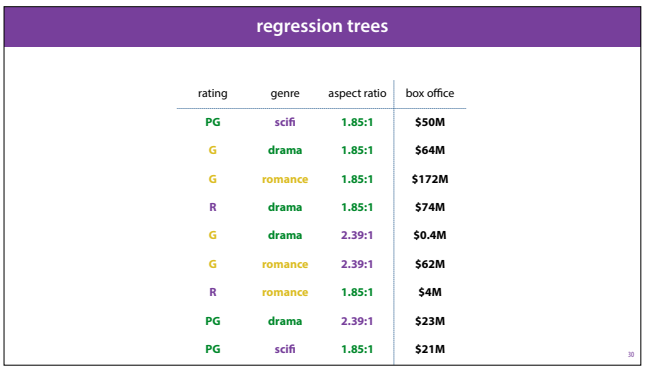




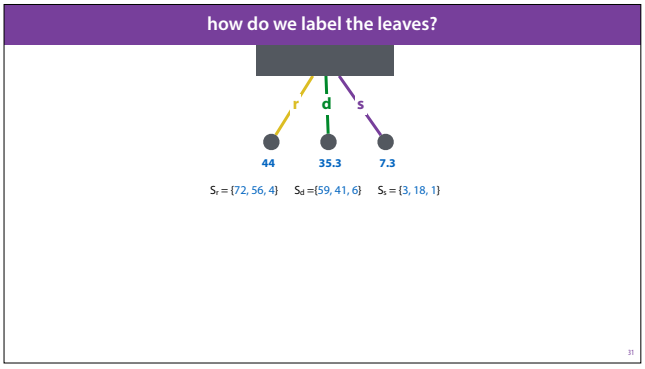
It's important to note that when we use a validation set to guide search, that we are using validation data to select our model. This means that if we are also using a validation set, for instance to select whether we'll use a kNN classifier or a decision tree, the pruning should happen on a withheld part of the training data, and not on the same validation data that we use to do our hyperparameter selection.

To see why, imagine what happens when we train our final model (supposing that we've selected a decision tree). During training, we can't use the test set to do our pruning. We can only see the test set when we've decided what our final model is going to be. The first train/validation split should simulate this,situation, so we can't use the orange validation set for controlling search.

This also goes for *early stopping* in neural networks

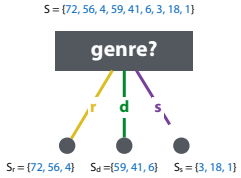


We've seen classification trees that use numerical feature, but what if the target label is numerical? In this case, the model is called a *regression* tree.



We can label the leaves with the mean or the median of the training instances in the resulting segment.

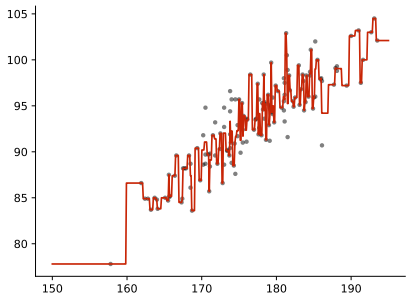
what do we use instead of entropy?



$$I_S(V) = \mathrm{Var}(S) - \sum_i \frac{|S_i|}{|S|}\mathrm{Var}(S_i)$$

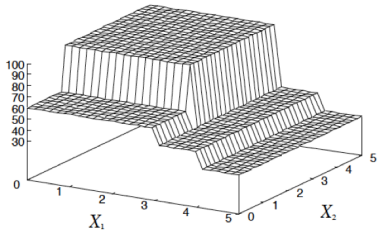
32

We can't compute entropy over the target values because most likely, they'll all be different. However variance measures a very similar property: the bigger the spread in the set of output labels, the less certain we are about what the value of the leaf node should be. The best split results in a large reduction of average variance over the created segments.



33

Here's what a regression tree looks like over one numeric feature.



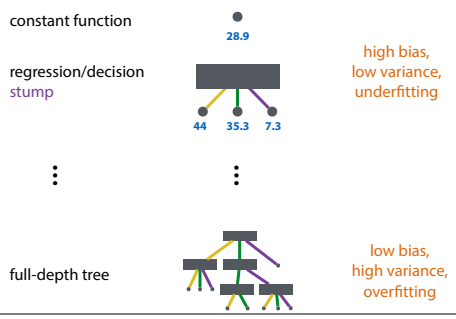
source: http://healthcare-economist.com/2015/11/16/what-are-regression-trees/

34

And here's what it looks like for two numeric features.

---

generalization hierarchy



35

Tree models are a classic example of a model class that provides a **generalization hierarchy**. At one end, the model class provides both very simple, low capacity models like **constant models**, which output just one value for all instances (i.e. a tree without splits) and **stumps**, models that make just one split. These are low capacity models with high bias and low variance, that generalize a lot.

At the other end are full-depth tree models, which are very likely to memorise irrelevant details of the data, and overfit a lot.

Single decision trees are not very popular. To make them effective, we need to train many of them and combine them into a single model. These are called decision or regression **forests**, and they're an example of a **model ensemble**, which we'll discuss after the break.

Trees and ensembles
Part 3: Ensembling

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

Ensembling is the business of combining multiple models into one. The hope is that this gives you a model that is more than the sum of its parts

---

Ensembling

Combining multiple models into a more powerful model: an **ensemble**.

Combine predictions by :

- majority vote (classification)
- average (regression)
- weight by confidence, validation accuracy
- stacking

Your collection of models is called the ensemble. In the simplest form of ensembling, each model in the ensemble can be trained in isolation as you would normally train it.

After you train a bunch of separate models, you need to somehow **combine** their predictions. The simplest approach in classification is to take a majority vote among the ensemble. The simplest approach in regression is to average the outputs. You can also take a weighted vote, perhaps giving greater weight to the predictions of models that have greater confidence, or models that perform better on a held out validation set.

A more complex approach is **stacking.**

stacking

height	age	
181	46	m
181	50	m
166	44	f
171	38	f
152	36	f
156	40	f
167	40	f
170	45	m
178	50	m
191	50	m
166	38	f
164	42	f
178	44	m

M_1: m m f m f m f f m m m f m

M_2: m f f f m f f f m m f m m

M_3: f f f f m f f m m f m f m

Imagine that we have a simple dataset, that we train three models on. These could be any three models. For instance, the three different models we saw in the first lecture, three neural networks, with different initializations, or three kNN models with different values of k. Each of them gives us a prediction for every instance in our dataset.

stacking

height	age	M_1	M_2	M_3	
181	46	m	m	f	m
181	50	m	f	f	m
166	44	f	f	f	f
171	38	m	f	f	f
152	36	f	m	m	f
156	40	m	f	f	f
167	40	f	f	f	f
170	45	m	f	m	m
178	50	m	m	m	m
191	50	m	f	f	m
166	38	f	m	m	f
164	42	f	m	f	f
178	44	m	m	m	m

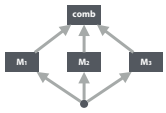
The idea of stacking is that we treat these predictions as features themselves, for another classifier. The simplest way to think of this is as adding these predictions to our dataset as columns.

We then train a new model, called a **combiner**, on this new, extended data. The combiner can choose to how to combine the "expert advice" of the original models, and it can even use the original features to learn which expert to listen to in which part of the feature space.

combiner: usually logistic regression

If NNs are used for the ensemble, the whole thing becomes one big neural network.

We can even refine the whole ensemble end-to-end using backpropagation



In general, the combiner is a simple, low-variance model like a logistic regression.

If we use stacking in combination with differentiable models like neural nets, then the stacked model is also fully differentiable, and can be finetuned end-to-end by backpropagation.

bias and variance

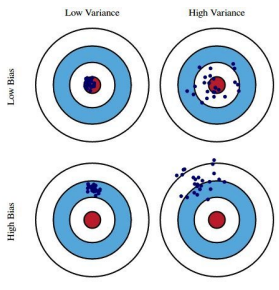


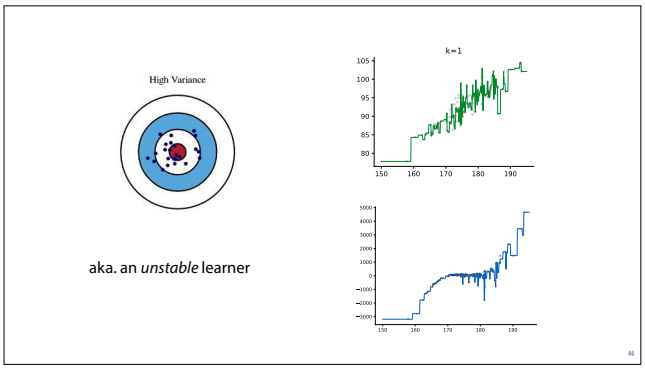
image source: **http://scott.fortmann-roe.com/docs/BiasVariance.html** (recommended reading)

That is all very ad-hoc. To get a better handle on exactly what we're trying to achieve with ensembling, we need to look back to bias and variance.

Here is a little visual reminder.

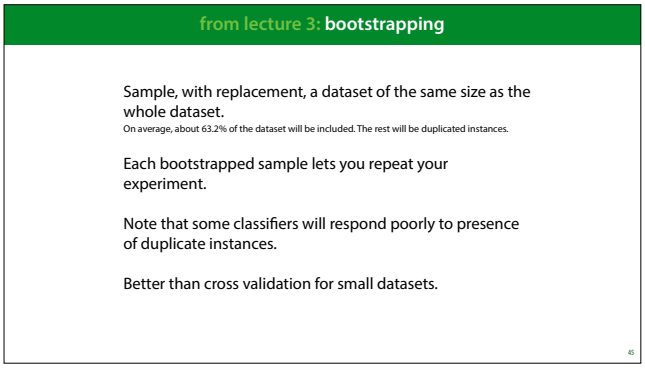
If we have many darts hitting close together, but far away from the bulls-eye, we have high bias. If the darts are spread out, but their average is the bulls-eye, we have high variance. It's important to remember that in this analogy sampling a dataset and training a model together counts as one dart. To get a second dart, we need a new dataset to train a new model on. That's not usually a luxury we have, so normally we can't be exactly sure whether our error is due to high bias or

high variance, but with tricks like resampling the data, we can often get a pretty good idea.



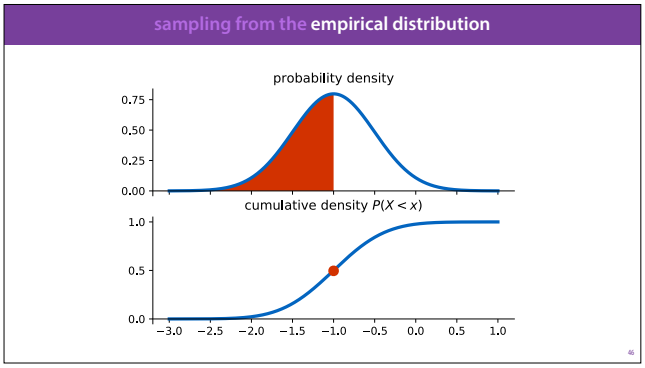
We'll start with learners, a model together with its search algorithm, that have **high variance**. We also call these **unstable learners.** These may get a good performance, but slight perturbations in the data can throw that performance off.

These are the kinds of models that tend to show overfitting, like kNN regression with a low k values, or a regression tree with no regularization. Bias and variance are only precisely defined for regression problems, but the basic intuition carries over to classification. An unstable learner is one that tends to overfit.



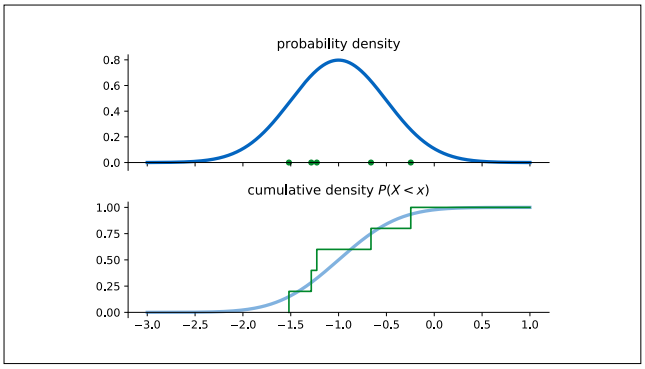
In lecture Methodology 2, we saw a method for simulating the sampling of multiple datasets form the source of our original data: **bootstrapping**. We can use bootstrapping both to get an idea of our bias/variance tradeoff and as a way to help us build an ensemble.

Before we do that, however, lets' see why bootstrapping works so well. It's more than just an intuitive trick. We can make precise exactly how it approximates our data distribution.

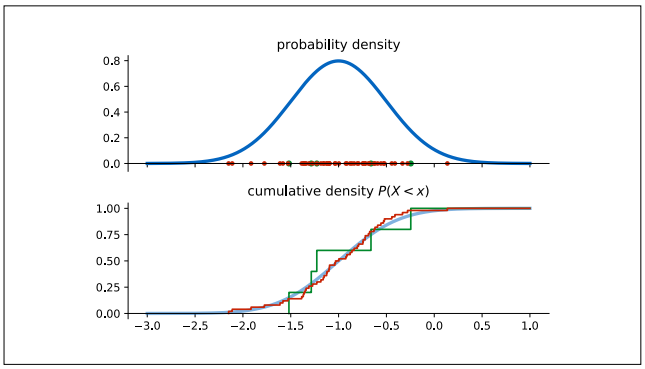


To make this clear, we'll imagine that we're sampling single scalars from a normal distribution.

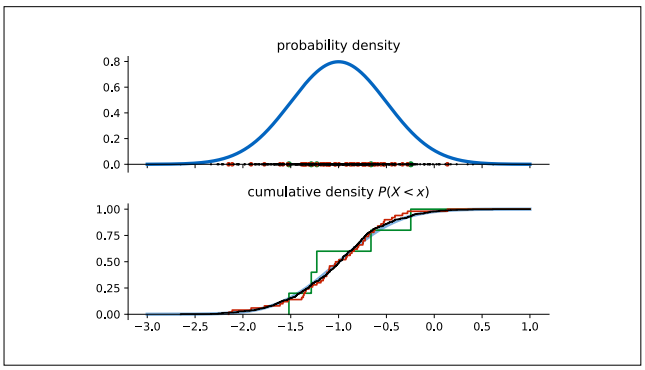
In this case, we can look at the **cumulative density function** (CDF). This tells us the probability of sampling a point below x. Note that this function returns a probability, not a density. It always groes from 0 to 1 over the domain of the probability distribution. Both the cumulative density function and the probability desnity function uniquely determine the probability distribution.



If we sample 5 points from the original normal distribution, and then re-sample one point from that dataset, we are essentially sampling from the green CDF. This is called the **empirical distribution**: the distribution we get by resampling one point in our dataset.

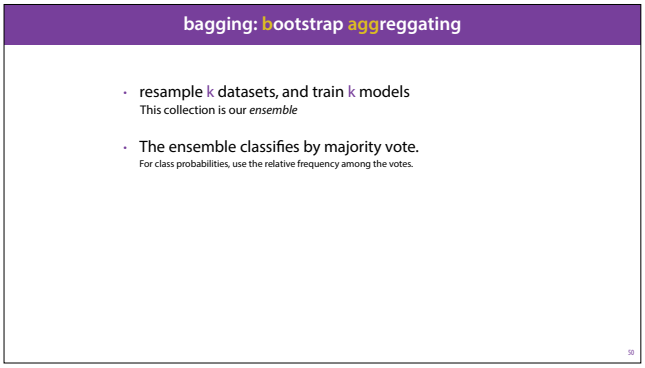


If we increase the size of the original data (to 50 points), we see that the empirical CDF becomes a better approximation of the true CDF



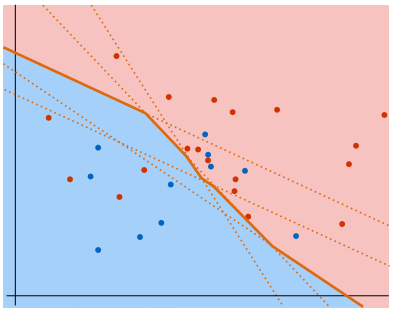
At 500 points, the empirical CDF and the original CDF are almost indistinguishable. This is why bootstrapping is often preferred over other resampling methods like cross validation. In this was we can show that a bootstrapped sample from a large dataset mimics the original data distribution.

This can help us to measure the bias and variance of a learner. It can also help us to build an ensemble.



This is called **bootstrap aggregating,** or **bagging** for short. We don't change the way we train the models in our ensemble, we just resample the data by bootstrapping.

This is most often done with classifiers. In that case, after the ensemble is trained, we simply take a majority vote to get the ensemble prediction. If we want class probabilities, we can use the relative frequencies of each class among the predictions.



51

Here's a simple example. We train a set of linear classifiers on bootstrap samples of our data. Each produces a slightly different linear decision boundary (indicates by a dotted line). We now build an ensemble that looks at what each of these classifiers says, and picks the majority class among those predictions. This gives us a piecewise linear decision boundary: every time two decision boundaries in the ensemble cross, the majority changes. So long as they don't cross, we end up following one of the original linear decision boundaries.

source: adapted from *Machine Learning* by Peter Flach, figure 11.1

---

random forests

Bagging with decision trees.

Subsample the data *and* the features for each model in the ensemble.

Reduces variance, few hyperparameters, easy to parallelise.

No reduction of bias.

52

One simple instantiation of bagging is the **random forest**. Here, we train a boostrapped ensemble of decision trees and for each we subsample both the instances, and the features we include (both the rowas and the columns of the data matrix).

---

summary: ensembling & bagging

Used in production (and competitions) to achieve extra performance on top of a particular model.

Never used in research. We *know* we can improve any model by boosting.
But it doesn't say anything about the model itself.

Can be expensive for big models. Ensembling with large deep learning models is rare in production.
Still happens in (Kaggle) competitions.

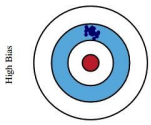
Bagging reduces *variance*, what about reducing *bias*?

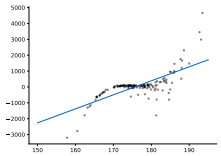
53

Trees and ensembles
Part 4: Boosting

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

the *hypothesis boosting* question





regression/decision stump

51

In the previous video, we saw the ensembling method of bagging in action.

Bagging helps for models with high variance and low bias. If we have the opposite, a learner with high bias and low variance, can we achieve the same? This is the **hypothesis boosting** question , where hypothesis is a synonym for a learner or model. Is there an ensembling method that allows us to create a series of models from a family with high bias, like linear models or decision stumps, and to create an ensemble that together has low bias (possibly at the expense of a higher variance).

---

boosting

w	height	age	class
1	181	46	male
1	181	50	male
1	166	44	female
1	171	38	female
1	152	36	female
1	156	40	female
1	167	40	female
1	170	45	male
1	178	50	male
1	191	50	male
1	166	38	female
1	164	42	female
1	178	44	male

56

Most boosting methods work by adding a **weight** to each instance in the data.

For each new model, we lower the weights of the points that the previous models got right, and increase the weight of the points that the previous models got wrong. We then train the next model to focus on this reweighed version of the data.

---

boosting (general idea)

train some classifier m_0

for t from 1 to k:

$M_{t-1}(x) = m_0 + a_1 m_1(x) + \ldots + a_{t-1} m_{t-1}(x)$
our ensemble so far

increase w for instances misclassified by M_{t-1}
normalise weights

train m_t on reweighted data, assign m_t a weight a_t
the better m_t, the higher a_t

M_k is our final model.

57

---

training on weighted data

Weighted loss function:

$$\text{loss}(\theta) = \sum_i w_i (f_\theta(\mathbf{x}_i) - t_i)^2$$

Or, resample your data, by the weights.
w_i determines how likely $\mathbf{x}_i$ is to end up in the resampled data.

58

How do we train on a weighted dataset? If we have a loss function, we can just make the sum over the loss of each instance a weighted sum. We saw examples of weighted maximum likelihood loss in the lecture on density estimation.

If our model isn't trained by an explicit loss function, like for instance the tree models in the first two videos, we can resample the dataset, and make the instance with higher weights more likely to be sampled. Think of this as a kind of weighted bootstrapping.

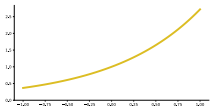
AdaBoost (for binary classifiers)

Weak learners m_t output -1 for Neg, 1 for Pos.
target values y_i are also -1 and 1

$M_{t-1}(x) = m_0 + a_1 m_1(x) + \ldots + a_{t-1} m_{t-1}(x)$
outputs a value between $-\infty$ and ∞, *sign* of the output is the class.

$M_t(x) = M_{t-1}(x) + a_t m_t(x)$

$$e_t^i = \exp(-y_i M_t(\mathbf{x}_i))$$
$$E_t = \sum_i e_t^i$$



There are a few ways to instantiate this general idea of boosting.

We'll first take a look at **AdaBoost** (which stands for *adaptive* boosting), which is a principled derivation for how to set the weights w_i, and and the model weight a_t.

We assume we've already trained the ensemble up to model m_{t-1}, giving us ensemble model M_{t-1}. We now need to make two choices: which model m_t to choose (or what loss to minimize when training this model) and which model weight a_t to assign it.

To do so, we first define the error (aka the loss) for the ensemble at step t, and then choose m_t and a_t to minimize this loss.

AdaBoost

$$\begin{aligned} e_t^i &= e^{-y_i M_t(\mathbf{x}_i)} \\ &= e^{-y_i (M_{t-1}(\mathbf{x}_i) + a_t m_t(\mathbf{x}_i))} \\ &= e^{-y_i M_{t-1}(\mathbf{x}_i)} \; e^{-y_i a_t m_t(\mathbf{x}_i)} \\ &= w_i e^{-y_i a_t m_t(\mathbf{x}_i)} \end{aligned}$$

We can rewrite the per-instance loss to separate the error cause by the ensemble so far, and the loss caused by our choice of model m_t. The first is a constant (since the ensemble up to m_{t-1} has already been chosen), which becomes the weight w_i of instance $\mathbf{x}_i$.

AdaBoost: choosing m_t

choose m_t to minimize

$$\begin{aligned} E_t &= \sum_i w_i e^{-y_i a_t m_t(\mathbf{x}_i)} \\ &= \sum_{\text{correct}} w_i e^{-a_t} + \sum_{\text{incorrect}} w_i e^{a_t} \\ &= e^{-a_t} \sum_{\text{correct}} w_i + e^{a_t} \sum_{\text{incorrect}} w_i \\ &= e^{-a_t} W_c + e^{a_t} W_i \\ &\rightarrow W_c + e^{2a_t} W_i \\ &= (W_c + W_i) + (e^{2a_t} - 1) W_i \end{aligned}$$

For the total loss, we can split the sum into the instance that are correctly classified and the instances that are incorrectly classified. This means the exponents become constants in the sum and can be move to the front. Line five is not an equal function, but minimising line four is the same as minimising line 5. In the last line, all the greyed out parts are constant with respect to m_t: W_c and W_i each depend on which classifier we choose, but their sum is just the sum of all the weights provided by the ensemble so far.

The take-away here is that choosing m_t to minimize the error E_t, consists of just minimising the sum of the weights of the instances misclassified by m_t.

AdaBoost: choosing a_t

choose a_t to minimize $\quad E_t = e^{-a_t} W_c + e^{a_t} W_i$

$$\begin{aligned} \frac{\partial E_t}{\partial a_t} &= \frac{\partial e^{-a_t}}{\partial a_t} W_c + \frac{\partial e^{a_t}}{\partial a_t} W_i \\ &= -e^{-a_t} W_c + e^{a_t} W_i \end{aligned}$$

$$a_t = \frac{1}{2} \ln \frac{W_c}{W_i}$$

for details, see: **AdaBoost and the Super Bowl of Classifiers**, Raúl Rojas.

To choose at, we just compute the derivative of the error wrt to a_t, set it to zero and solve for a_t.

Intuitively, the formula for a_t states that the better the proportion of correct to incorrect labelings, the more model t should weigh in the ensemble. The logarithm ensures a kind of diminishing return of weight: getting 11 instances correct instead of 10 had much more impact on the weight than getting 101 instances correct instead of 100.

AdaBoost

train some classifier m_0

for t from 1 to k:

$M_{t-1}(x) = m_0 + a_1 m_1(x) + \ldots + a_{t-1} m_{t-1}(x)$
our ensemble so far

$w_i = e^{-y_i M_{t-1}(x_i)}$

to choose m_t:
minimize sum of weights w_i of incorrect classifications

to choose a_t: $a_t = \frac{1}{2} \ln \frac{W_c}{W_i}$

M_k is our final model.

63

boosting

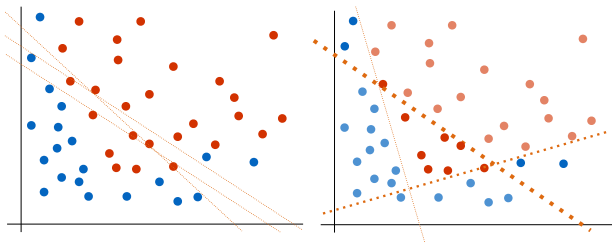
works even if M only classifies just slightly better than chance

- i.e. decision *stumps*

variants: AdaBoost, LogitBoost, BrownBoost

64

bagging vs boosting



65

If we visualise the learners, we can see clearly why boosting is so much more powerful than bagging. Since bagging works in parallel, every member of the ensemble will end up looking roughly the same, providing little variation in the ensemble. If we start with underfitting classifiers (like linear ones), the ensemble decision boundary doesn't look much different from the individual ones.

In boosting, since each learner is trained in sequence, based on what the previous learners did, we get much more variation, giving us a combined decision boundary that can be much more powerful than what the original decision boundaries looked like.

Adapted from Machine Learning by Peter Flach, figure 11.2

gradient boosting

Boosting for *regression* models

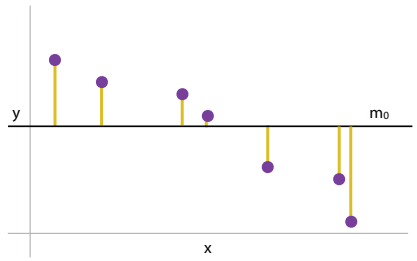
Main intuition: fit the next model to the residuals of the current ensemble.

66

When we want to boost regression models, we can look to **gradient boosting**. The idea here is that we don't reweight the dataset, but instead, we look at the residuals of the ensemble to far, and try to train a model to predict those. If we can keep doing this succesfully, we can eventually subtract all residuals to get a perfect prediction.

---

gradient boosting

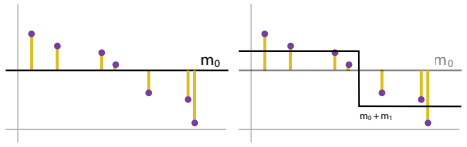


67

To illustrate, let's start with a constant model. We'll minimize squared error, so the optimal constant is the mean of the data.

---

gradient boosting (informally)



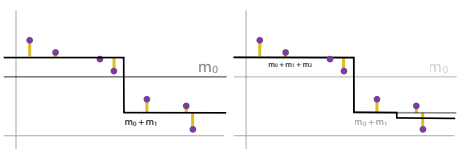
m_1 predicts the residuals of m_0

68

We take the residuals of the previous model, and train a new model, m1, to predict the residuals. The new *ensemble model* adds these predictions to the predictions of the first.

---

gradient boosting (informally)



69

This ensemble model has new residuals, which we can then train another model m2 to predict, add that to the model. and so on.

gradient boosting (for least squares loss)

initial model $M_0(\mathbf{x}) = c$

for t from 1 to k:

  for all i: $r_i \leftarrow M_{t-1}(\mathbf{x}_i) - y_i$
  r_i are the residuals of the ensemble so far

  fit model m_t to dataset $\{\mathbf{x}_i, r_i\}$

  $M_t(\mathbf{x}) = M_{t-1}(\mathbf{x}) + \gamma_t m_t(\mathbf{x})$
  γ_t optimise through *line search*, or just slowly decay

This is the algorithm in detail. We start with an ensemble model consisting of only a constant predictor. At each iteration, we compute the residuals for the current ensemble model, and add a predictor for the residuals to the ensemble. The new model is added with a weight gamma, which helps us if the new model happens to fit the residuals poorly.

In constrast to adaboost, there is no principled way to set the weights. We simply search for a good value by optimization, or slowly decay the weights.

why *gradient* boosting?

Imagine a model which simply stores a single output for each instance

$$f_w(\mathbf{x}_i) = w_i$$

$$\frac{\partial \frac{1}{2}(f(\mathbf{x}_i) - y_i)^2}{\partial w_i} = w_i - y_i$$

To see why we call it gradient boosting, imagine a model which simply stores a single value w_i which is the value it will predict for instance $\mathbf{x}_i$. This is a perfectly overfitting model, of course, and we cannot expect it to learn anything, but we can think of its negative gradient as a kind of ideal direction for where we want a more realistic model to go. If we could control all the outputs of our model for all instances individually, we would want to follow this negative gradient to minimize the loss.

We can't, of course, in a realistic model, changing one parameter changes multiple predictions at the same time, but the ideally, this is the gradient on our "output space" that we want to approaximate.

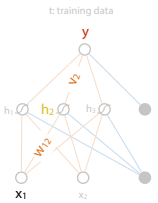
If our loss function is the mean squared error loss, then the gradient for this model are the residuals

why *gradient* boosting?

$$r_i = \frac{\partial \frac{1}{2}(M(x_i) - y_i)^2}{\partial M(x_i)} = M(x_i) - y_i$$

In other words, the residuals are the gradient of the model loss for instance i with respect to the output for instance i.

By adding the residuals to the previous model, we are performing a kind of gradient descent for models that don't support it (like regression trees). We are training a model to predict where we'd like to be after one step of gradient descent, and adding that model to our ensemble. the ensemble is following the approximate gradient in output space.



$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_y$$

$$h_2 = \frac{1}{1 + \exp(-h_2')}$$

$$h_2' = w_{12} x_1 + w_{22} x_2 + b_2$$

$$\frac{\partial l}{\partial w_{12}} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial h_2'} \frac{\partial h_2'}{\partial w_{12}}$$

$$= 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

$$w_{12} \leftarrow w_{12} - \eta \cdot 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

71

Compare this to what we saw in backpropagation. There, we also work out the derivative of the loss with respect to the network output y (we called it a *local derivative* in this context).

We then took this derivative and backpropagated it down the network to work out derivatives for the weights. We can think of gradient boosting as a way of accomplishing this for models where backpropagation isn't possible. Instead, we train a model to predict the effect of the gradient update step on the output space, and we bolt that model onto our original.

---

pseudo-residuals

$$r_i = \frac{\partial \text{loss}(M(x_i), y_i)}{\partial M(x_i)}$$

74

The benefit of this perspective is that it allows us to generalise the idea of gradient descent to other loss functions. We can simply work out this derivative, and train the next model in our ensemble to predict it. We call this a **pseudo-residual**.

The resulting value isn't as intuitive as a proper residual, but training the next model to predict the pseudo residuals works just as well to minimize the loss.

---

for instance: MAE loss

$$\text{loss}(m_i, y_i) = |m_i - y_i|$$

$$r_i = \frac{\partial |m_i - y_i|}{\partial m_i}$$

$$= \frac{\partial |m_i - y_i|}{\partial m_i - y_i} \frac{\partial m_i - y_i}{\partial m_i}$$

$$= \text{sign}(m_i - y_i)$$

75

For instance, here is how it works for the **mean absolute error** loss (also known as the L1 loss): the absolute value of the difference between output and target. As we've seen before, minimizing absolute errors instead of squared errors, leads to gradients c omposed of the sign, and the median as a minimizer.

Therefore, if we want to use gradient boosting to minimize the MAE loss, we should train the next model in our ensemble to predict only the *sign*, -1 or +1, of the residuals of the ensemble so far, and add that to the current predictions.

---

gradient boosting vs AdaBoost

Gradient boosting

`sklearn.ensemble.GradientBoostingClassifier` or `pip install xgboost`

Each model fits the (pseudo) residuals of the current ensemble.

The ensemble optimises a global loss.

Even if the individual models don't optimise a well-defined loss

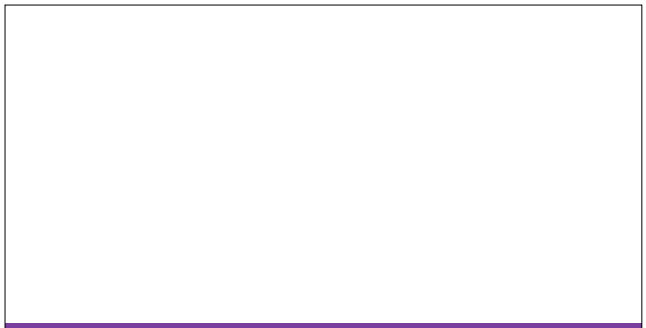
AdaBoost

`sklearn.ensemble.AdaBoostClassifier`

Each model fits a reweighted dataset.

Each model refines its own reweighted loss.

76



mlcourse@peterbloem.nl